Fine grain software pipelining of non-vectorizable nested loops

Ki-chang Kim and Alexandru Nicolau
Department of Information and Computer Science
University of California, Irvine
Irvine, CA. 92717

Technical Report #91-03
January 1991

Fine grain software pipelining of non-vectorizable nested loops

Abstract

This paper presents a new technique to parallelize nested loops at the statement level. It transforms sequential nested loops, either vectorizable or not, into parallel ones. Previously, the wavefront method was used to parallelize non-vectorizable nested loops. However, in order to reduce the complexity of parallelization, the wavefront method regards an iteration as an unbreakable scheduling unit and draws parallelism through iteration overlapping. Our technique takes a statement rather than an iteration as the scheduling unit and exploits parallelism by overlapping the statements in all dimensions. In this paper, we show how this finer grain parallelization can be achieved with reasonable computational complexity, and the effectiveness of the resulting method in exploiting parallelism.

1 Introduction

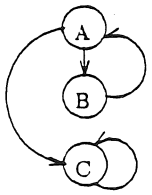
Loops are some of the richest program constructs where parallelism is available. Especially as the nest depth of the loop increases, the time that the CPU spends in it sharply climbs up. Many vectorization techniques have been developed to exploit the parallelism hidden in this construct [Kenn80] [KKLW80][Wolf82][AlKe87]. For loops[1] which are not vectorizable, however, the general technique is the wavefront method [Mura71][Lamp74][Kuhn80].

An elegant way of implementing the wavefront method through the combination of loop skewing and loop interchanging[AlKe84] is shown in [Wolf89]. [Bane90][LaWo90] show recent developments in this direction. But since in the wavefront method the unit of scheduling is an iteration, the parallelism *inside* iterations is not utilized. Each iteration is regarded as an atomic computational unit and executed by a single processor sequentially. This approach is useful to reduce the parallelizing complexity for nested loops, but it also reduces the amount of parallelism exploitable. Integrating fine and coarse grain parallelism is particularly important in the light of the growing popularity of superscalar and VLIW machines such as the i860, i960, or the IBM R6000. In this context, machines (such as the Touch Stone project) are emerging that makes exploitation of parallelism at all levels critical.

[1] In this paper, we use "loops" and "nested loops" interchangeably to denote nested loops. When a "loop" indicates a "single loop", we will stress it explicitly unless it is clear from the context.

For $i = 0$ to $N - 1$
A: A[i] = f(B[i-1])
B: B[i] = g(A[i])
C: C[i] = h(A[i], C[i-1])
Endfor

(a) The source code and its dependence graph.



For $i = 0$ to $N - 1$
A: A[i] = f(B[i-1])
B: B[i] = g(A[i]) C: C[i] = h(A[i],C[i-1])
Endfor

(b) Optmally parallelized form.

cycle	schedule
0	A0
1	B0 C0
2	A1
3	B1 C1
4	A2
5	B2 C2
6	. . . .

(c) ASAP schedule

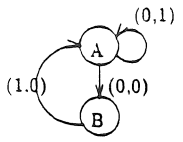
Figure 1: Optimal loop scheduling – one dimensional case.

Parallelizing loops at the fine grain level (statement level) has been pursued by other numerous researchers[Fish79][Nico85][GrLa86] [CCK87][DiXi87][Lam87][AiNi88]. For one dimensional loops, there exists an optimal solution[AiNi88]. Figure 1(a)-(b) shows an example loop and its parallelized form at the statement level. The parallelized form can be obtained by the following process. We unwind the loop repeatedly while scheduling each statement instance at the earliest cycle it can be executed (ASAP schedule) until a pattern is detected in the schedule. [2] Figure 1(c) shows this scheduling process, where the pattern is enclosed with a box. Then, we replace the original loop body with this pattern. The schedule obtained this way is known to be optimal[AiNi88]. However, previous attempts[Nico87] to expose fine grain parallelism in nested loops have not been totally satisfactory. Loop Quantization[Nico87] computes the amount of unwinding for nested loops, but does not maximize parallelism.

In this paper, we extend the approach in [AiNi88] to the n dimensional case to achieve fine grain parallelization of nested loops. Finding an optimal schedule for the n dimensional case is an open problem. For some cases, we can easily find optimal schedules. Figure 2(a) shows

[2] Throughout this paper, "schedule" means a static reordering of the statements by the compiler. This schedule is then executed *as is* , i.e. with the order of the statements in the schedule being preserved. This corresponds to the VLIW/superscalar model, and can be explicitly enforced in other parallel machines.

```
For i1 = 0 to N1 - 1
  For i2 = 0 to N2 - 1
    A:A(i1,i2)=f(A(i1,i2-1),B(i1-1,i2))
    B:B(i1,i2)=g(A(i1,i2))
  Endfor
Endfor
```



(a) The source and its dependence graph. In the graph, each edge is associated with a dependence distance vector.

```
for t = 0 to (N1 - 1)2 + (N2 - 1) + 1
    forall i1 = L1 to U1
        forall i2 = L2 to U2
            case Max(0, t - 2i1 - i2) is
                0: A_{i1,i2} = f(A_{i1,i2-1}, B_{i1-1,i2})
                1: B_{i1,i2} = g(A_{i1,i2})
            endcase
        endfor
    endfor
endfor
where L1 = Max(0, ⌈(t - N2)/2⌉),
      U1 = Min(N1 - 1, ⌊t/2⌋),
      L2 = Max(0, t - 2i1 - 1),
      U2 = Min(N2 - 1, t - i1).
```

(c) The parallel form.

cycle	schedule
0	A00
1	B00 A01
2	B01 A02 A10
3	B02 A03 B10 A11
4	B03 A04 B11 A12 A20
5	B04 A05 B12 A13 B20 A21
6	B05 B13 A14 B21 A22
7	B14 A15 B22 A23
8	B15 B23
9	

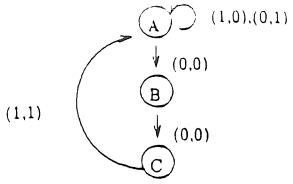
(b) ASAP schedule

Figure 2: Optimal loop scheduling – 2 dimensional case.

such an example. In the figure, the dependence edges are annotated with the dependence distance vectors. Thus, $(0,0)$ means an *in-loop* (non loop carried) dependence, $(0,1)$ means the dependence carried by loop i_2, etc. The ASAP schedule of this loop after some number of unwindings is given in Figure 2(b). We observe that the delay of statements A and B along the i_1 dimension is always 2, while along the i_2 dimension it is always 1. For example, $A(i_1, i_2)$ can be executed only 1 cycle after $A(i_1, i_2 - 1)$ and 2 cycles after $A(i_1 - 1, i_2)$. A similar argument applies for statement B. Because of this regularity, we can parallelize the original loop as in Figure 2(c).[3] The parallelized loop is exposing *all* statement level parallelism in this loop. The reader can verify this by following its execution several steps. At each step, the loop correctly executes all statments that can be done as soon as possible.

However, in general, when we schedule all the statement instances ASAP, we do not necessarily see a fixed delay pattern emerge as in the previous example. One example is given in Figure 3. The delay pattern is given in Figure 3(c). In the figure, $d_1(i_1, i_2)$ is the delay along the first dimension at iteration (i_1, i_2), and $d_2(i_1, i_2)$ the delay along the second dimension at iteration (i_1, i_2). For example, the delay between A_{i_1,i_2} and A_{i_1+1,i_2} is represented by $d_1(i_1, i_2)$. Similarly, the delay between A_{i_1,i_2} and A_{i_1,i_2+1} is represented by

[3]The details of this transformation are in Section 2.2.4.



(a) The dependence graph

cycle	schedule
0	A00
1	B00 A01 A10
2	C00 B01 A02 B10 A20
3	C01 B02 A03 C10 A11 B20
4	C02 B03 B11 A12 C20 A21
5	C03 C11 B12 A13 B21
6	C12 B13 C21 A22
7	C13 B22 A23
8	C22 B23
9	C23

(b) ASAP schedule

$$d_1(i_1, i_2) = \begin{cases} 2 & \text{if } i_1 < i_2 \\ 1 & \text{if } i_1 \geq i_2 \end{cases}$$

$$d_2(i_1, i_2) = \begin{cases} 2 & \text{if } i_1 > i_2 \\ 1 & \text{if } i_1 \leq i_2 \end{cases}$$

(c) Delay pattern.

Figure 3: A 2-dimensional loop in which the delays are not constant.

$d_2(i_1, i_2)$. For both dimensions, the delays are not constant; they could be 1 or 2 depending on the values of i_1 and i_2. It is an open problem whether we can optimally parallelize loops whose delays are not constants but functions of index variables, as in this example. Furthermore, as will be shown in Section 2, an optimal solution would require knowing the exact bound of all the loops at compile time; since this information is not usually available, an optimal solution is, in general, only of theoretical interest.

Instead of trying to parallelize loops optimally for all cases of delay patterns, we simply force the delays to be constant. So, first we select a set of delays, and then parallelize the loop based on it.[4] How to find an efficient set of delays and how to parallelize the loop based on it are the two main topics in Section 2. We compare our method with the current wavefront methods[WolfS9][Bane90][LaWo90] in Section 3.

[4]The problem of finding an efficient set of delays was also mentioned in [Cytr84]. In Section 2.2.2, we compare our method to the one described there.

2 Fine-grain Parallelization of Nested Loops

2.1 Definitions

Before we present the details of our technique, we need to define several terms that will be used throughout the paper. We assume the loop is tightly-nested with dimensionality n; otherwise, the loop can be converted into such a canonical form using the techniques in [AbuS78].[5] Also, we assume all statements take one unit of time. This assumption is for the ease of explanation *only*. Our techniques in this paper extend to the general case.

We follow the standard definitions of the dependence graph of a nested loop as in [Padu79], with nodes in the graph representing statements of the loop body, and edges representing dependences.[6] A node will be denoted by x_j, for the j_{th} node, and an edge by $e^i_{(s,d)}$, for the i_{th} edge whose dependence comes from node x_s (source) to node x_d (destination). We define the *iteration space*, again following the standard usage. An iteration in this space will be denoted by $I(i_1, \ldots, i_n)$, where "$0 \leq i_1 \leq N_1 - 1, \ldots, 0 \leq i_n \leq N_n - 1$ with $N_1, \ldots, N_n$ being the loop bounds."[7] The execution order of this space is lexicographic with i_1 being the outermost loop. We will call $(i_1, \ldots, i_n)$ the coordinate of this iteration. The instance of a node x_j at iteration $I(i_1, \ldots, i_n)$ will be denoted by $x_j^{I(i_1,\ldots,i_n)}$.

For each edge, e^i, we associate a *dependence distance vector* $\mathbf{v}_i = (v_{i1}, v_{i2}, \ldots, v_{ij}, \ldots, v_{in})$, where v_{ij} represents the dependence distance in the j_{th} dimension of the loop by the dependence edge e^i. For the ease of explanation, we assume the dependence distance is a constant integer. Our algorithms in this paper extends to the general case in which the dependence distance is expressed as a set of integers as in [LaWo90].

Definition 1 *A dependence distance vector,* $\mathbf{v}_j$, *is* legal *only if* $(i_1 + v_{j1}, \ldots, i_n + v_{jn}) \geq (i_1, \ldots, i_n)$, *lexicographically, for all* $(i_1, \ldots, i_n)$ *in the iteration space.*

The following lemma will be used to prove the correctness of Algorithm *dvector* in Section 2.2.1.

[5] In fact, our technique can be easily adapted to deal directly with non-strictly nested (loosely-nested) loops. However, in this paper, we only deal with tightly-nested loops.

[6] We will use "nodes" and "statements" interchangeably in this paper depending on whether we want to emphasize their place in the dependency graph or the schedule.

[7] We assume the loop is normalized by the standard method as in [Bane88].

Lemma 1 *The left-most non-zero element of a legal dependence distance vector* $\mathbf{v}_j$ *is positive.*

Proof. If the left-most non-zero element of $\mathbf{v}_j$ is a negative number, $\mathbf{v}_j$ is not a legal dependence distance vector from Definition 1. Q.E.D.

Definition 2 *The* shape *of a loop is an augmented dependence graph in which each node is assigned an integer, called an* $OFFSET$ *value. The* $OFFSET$ *value of node* x_j, $OFFSET(x_j)$, *is the partial ordering number of* x_j *for some partial ordering of the nodes that satisfies all* in-loop dependences.

Examples of the *shape* of a loop are shown in Figure 5(a)-(c). The *shape* of a loop can be thought of as an execution schedule.

For a *shape*, we define a *delay vector.*

Definition 3 *A* delay vector *of a shape is* $(d_1, \ldots, d_n)$, *where* $d_j \geq 0\ \forall j$, *which satisfies the following inequalities.*

$$\begin{aligned}
(v_{11}d_1 + v_{12}d_2 + \ldots + v_{1n}d_n) &\geq c_1 \\
(v_{21}d_1 + v_{22}d_2 + \ldots + v_{2n}d_n) &\geq c_2 \\
&\ldots\ldots \\
(v_{l1}d_1 + v_{l2}d_2 + \ldots + v_{ln}d_n) &\geq c_l, \qquad (1)
\end{aligned}$$

where c_i *is* $OFFSET(x_s) - OFFSET(x_d) + 1$ *for the* i_{th} loop carried dependence edge, $e^i_{(s,d)}$, $(v_{i1}, \ldots, v_{in})$ *is its dependence distance vector, and* l *is the number of* loop carried dependence *edges.*

We will refer to the matrix v_{ij} as $\mathbf{v}$ matix or *dependence distance matrix.* Also, we will use $\mathbf{c}$ to denote vector $(c_1, \ldots, c_l)$. $\mathbf{v}_i$ will refer to the i_{th} row of $\mathbf{v}$, and $\mathbf{v}^j$ to the j_{th} column of $\mathbf{v}$. Also we will use $\mathbf{v}^J_I$ to denote the submatrix of $\mathbf{v}$ whose rows and columns are specified by two integer sets, I and J. $\mathbf{c}_I$ is similarly defined as the subset of $\mathbf{c}$ whose elements are specified by set I.

The following lemma will also be used to prove the correctness of Algorithm *dvector* in Section 2.2.1.

Lemma 2 *If the j_{th} column of the dependence distance matrix* **v** *contains only positive elements, then there exist a delay vector whose elements are all zero's except at the j_{th} position.*

Proof. Substitute a delay vector $(0, \ldots, 0, d_j, 0, \ldots, 0)$, where $d_j \geq MAX(\lceil c_1/v_{1j} \rceil, \ldots, \lceil c_l/v_{lj} \rceil)$, into Inequalities 1. We obtain

$v_{1j}d_j \geq c_1$

$v_{2j}d_j \geq c_2$

...

$v_{lj}d_j \geq c_l$.

The inequalities are all satisfied automatically since $v_{ij} \geq 1$ for all $i \in [1..l]$. Q.E.D.

Definition 4 *For a node instance in a* shape, $x_j^{I(i_1,\ldots,i_n)}$, *we define a function $CYCLE$ as*

$$CYCLE(x_j^{I(i_1,\ldots,i_n)}) = i_1 \times d_1 + i_2 \times d_2 + \ldots + i_n \times d_n + OFFSET(x_j), \qquad (2)$$

where $(d_1, \ldots, d_n)$ is the delay vector of the shape.

In our execution model, each node instance, x, is executed at cycle $CYCLE(x)$.[8] Since the parallel execution time of the loop is the same as the completion time of the last iteration, whose coordinate is $(N_1-1, N_2-1, \ldots, N_n-1)$, we can compute *partime*, the parallel execution time of the loop, as follows.

$$partime = (N_1 - 1) \times d_1 + \ldots + (N_n - 1) \times d_n + L, \qquad (3)$$

where L is the execution time of a single iteration of the parallelized loop in our execution model.

A delay vector is *optimal* if it minimizes *partime* in *Equation* 3.

2.2 Our approach

To derive a parallel execution schedule for the nested loops, we first fix the shape of the loop, which means we fix the $OFFSET$ values for all statements (Section 2.2.3). Then we calculate

[8] The delay vector should be chosen to make this execution model valid. In Lemma 3, we prove the delay vector chosen by Algorithm *dvector* validates this model.

the delay vector based on the $OFFSET$ values (Section 2.2.1). And then, we transform the loop into a parallel one using this delay vector (Section 2.2.4).

2.2.1 Calculating the delay vector

Before discussing the problem of determining $OFFSET$ values, we present a method to calculate the delay vector. Assuming that the $OFFSET$ values for all statements are known, the problem of finding an optimal delay vector can be reduced to the following integer programming problem:

> Compute $(d_1, d_2, \ldots, d_n)$ such that it satisfies Inequalities 1 and minimizes *partime* in Equation 3.

Since some of the loop bounds are frequently unknown at compile time, solving this problem optimally is not always possible even if we are willing to incur the cost of integer programming. Instead, we take a simple approach which maximizes the number of zero elements in the delay vector. Note that if there is a zero element in the delay vector, the loop is vectorizable for that dimension.

From Lemma 2, if one of the columns of the **v** matrix (the dependence distance matrix) has only positive components, then there exists a delay vector in which all elements except one are zero's. By replacing those zero delays in Inequalities 1, the non-zero element can be easily calculated. If there is no positive column in **v** (a column whose components are all positive), we build a submatrix of **v** by removing all rows which start with a non-zero element, as well as the first column. For this submatrix we calculate a partial delay vector (the first element of the delay vector is not computed) as follows. If one of the columns in this submatrix is positive, we can compute the partial delay vector easily (see above). If no positive columns exist in the submatrix again, we apply the same process recursively: extract another new submatrix by removing all rows starting with non-zero elements and the first column from the current submatrix; search for a positive column in it; if there is one, we are done, otherwise repeat; and so on. Note that for any submatrix extracted during the recursion, the first column cannot contain any negative elements. This is because when we build a new submatrix, we extract only those rows starting with zero elements from

Algorithm. dvector.
Input. $\mathbf{v}_I^J$ and $\mathbf{c}_I$. I and J are two integer sets with cardinalities $IMAX$ and $JMAX$.
Output. $\mathbf{d}_J$
Method.

1. If all the elements of the column vector $\mathbf{v}^{J_j}$ are positive,
   $d_{J_j} = MAX(\lceil c_{I_1}/v_{I_1J_j} \rceil, \ldots, \lceil c_{I_{IMAX}}/v_{I_{IMAX}J_j} \rceil)$
   $d_x = 0, \forall x \in J$ except J_j
   Else
   $I' = \{I_k \text{ s.t. } v_{I_kJ_1} = 0\}$
   $J' = \{J_k, \forall k \neq 1\}$
   call algorithm dvector with $\mathbf{v}_{I'}^{J'}$ and $\mathbf{c}_{I'}$ to calculate $\mathbf{d}_{J'}$.
   $d_{J_j} = MAX(\lceil c'_{I_1}/v_{I_1J_1} \rceil, \ldots, \lceil c'_{I_{IMAX}}/v_{I_{IMAX}J_1} \rceil)$.

Figure 4: Algorithm to compute a delay vector.

the previous submatrix, and the first non-zero element in these rows should be positive by Lemma 1.

The exact algorithm is shown in Figure 4, where c'_{I_i} is $c_{I_i} - \sum_{x \in (J-J_1)} v_{I_ix} d_x$. Initially the algorithm starts with $I = (1, 2, \ldots, l)$ and $J = (1, 2, \ldots, n)$, where l is the number of *loop carried dependences* in the loop, and n is its dimensionality.

The time complexity of algorithm *dvector* is $O(l^2 n)$, derived from the recursive equation,

$$D_l = D_{l-1} + O(l) + O(ln),$$

where D_l is the time *dvector* takes to compute the delay vector for $\mathbf{v}$ with l rows. Solving this recursion, we get $O(l^2 n)$ time complexity. Note that this is the worst case, as the solution is often found much earlier in practice.

We now prove that the delay vector computed in this way satisfies all dependences in the original loop when all the statement instances are scheduled at the cycles specified by Equation 2. Conceptually the delay vector gives the legal start of an iteration. The individual statements in the iteration are offset from this start by an amount ($OFFSET$) computed from a partial ordering of statements that satisfies the in-loop (non-loop-carried) dependences.

Lemma 3 *For a loop whose delay vector is obtained as above, scheduling all instances of statements at the cycles computed by Equation 2 satisfies all the dependences.*

Proof. We take an arbitrary instance of an arbitrary edge and prove its dependence is satisfied. Call the edge $e_{(s,d)}$. Suppose its source statement-instance happen to be in $I(i_1, \ldots, i_n)$. Also let its dependence vector be $(v_1, \ldots, v_n)$. From Equation 2,

$CYCLE(\text{source statement-instance}) = CS$
$= i_1 \times d_1 + i_2 \times d_2 + \ldots + i_n \times d_n + OFFSET(x_s)$
$CYCLE(\text{destination statement-instance}) = CD$
$= (i_1 + v_1) \times d_1 + (i_2 + v_2) \times d_2 + \ldots + (i_n + v_n) \times d_n + OFFSET(x_d).$

So, $CD - CS = v_1 \times d_1 + \ldots + v_n \times d_n + OFFSET(x_d) - OFFSET(x_s)$. We need to prove that this value is greater than or equal to 1 for any instance of the edge (because we assume all statements take 1 unit of time). For in-loop dependences, $v_i = 0, \forall i$, and $OFFSET(x_d) - OFFSET(x_s) \geq 1$ (from Definition 2); therefore, $CD - CS \geq 1$. For loop-carried dependences, $CD - CS \geq 1$ from Inequalities 1. Q.E.D.

2.2.2 Comparing with DOACROSS method

[Cytr84] suggests an algorithm to find an efficient delay vector for multi-dimensional DOACROSS. However, since that method is intended for general loosely nested loops, it is not as efficient as ours which assumes the loop is tightly nested. Basically, to find d_j, the DOACROSS method has to look at all inequalities in Inequalities 1, while our method looks at only a subset of them – only those inequalities which satisfy $v_{r1} = v_{r2} = \ldots = v_{r,j-1} = 0$, and $v_{rj} > 0$. Since d_j is the smallest integer that satisfies the corresponding set of inequalities in both cases, our solution is always smaller than or equal to the solution by the DOACROSS method.

Another difference is the choice of $OFFSET$ values. As can be seen from Inequalities 1, it is important to decrease the values of $c_1, \ldots, c_l$ to minimize the delay vector and thus increase parallelism. Since the elements of the delay vector are dependent on the $OFFSET$ values, we need to choose the right set of $OFFSET$ values, which means we need to select the right partial ordering of the statements. In the DOACROSS method, the original order of the statements is not disturbed, thus there is no search for the right set of $OFFSET$ values. The reordering problem mentioned in [Cytr86] and [MuSi87] is also different from our problem in that it is a total ordering problem which has been shown to be NP-hard[Cytr84]. In the next section, we present our method to find an efficient partial ordering.

2.2.3 Fixing $OFFSET$ values

The problem of finding the right set of $OFFSET$ values is the problem of finding the right partial ordering of the statements in the loop, or finding the right *shape* of the loop. The optimal *shape* would be one that the delay vector computed from this *shape* minimizes the parallel execution time in Equation 3. However, computing a delay vector that minimizes the parallel execution time is not always possible since the loop bounds are frequently unknown at compile time, as mentioned in Section 2.2.1; therefore, finding the optimal *shape* is also impossible in these cases. Our strategy is to find a *shape* which produces as many as possibly zero elements in the delay vector.

One simple way is to take the first iteration of the loop, execute it sequentially, and let the starting cycle of each statement be its $OFFSET$ value. Let's call this the *original-order method.* A variation of this method is to take the first iteration and schedule it ASAP, letting the starting cycle of each statement be its $OFFSET$ value. This will be referred to as the *compacted-order method.* Neither of them is satisfactory. We take an approach derived from fine grain scheduling. We unwind the loop a finite number of times for each dimension, schedule all statement instances in it ASAP, and compute the starting cycles of the statement instances once a repeating pattern is found. Such a pattern will often occur naturally, and in those cases it in fact expresses the optimal parallelism in the loop; when the pattern does not occur on its own, we simply force it heuristically after a finit number of iterations. Let the iteration at which a pattern first emerges or at which we force a pattern be I and the minimum of the starting cycles of the statement-instances in that iteration be $CYCLE(x_s^I)$, that is statement x_s starts first in iteration I. Then, the $OFFSET$ value of statement x_i is computeded by $CYCLE(x_i^I) - CYCLE(x_s^I)$.[9] This method will be referred to as the *scheduled-order method.*

Figure 5(a)-(c) shows a sample loop and the three different sets of $OFFSET$ values for it. In the figure, we also calculated the delay vector for each case using Algorithm *dvector.*

The merit of the *scheduled-order method* is that it rearranges the statements to expose hidden parallelism. For example, in Figure 5(a)-(c), we don't get vectorizable dimensions

[9] Note that we perform the unwinding *only* to compute the $OFFSET$ values; we do not replace the loop body with the unwound loop body.

that each dimension defines its own hyperplanes. For example, the i_1 dimension defines N_1 $(n-1)$-dimensional hyperplanes in an n-dimensional iteration space. The exact form of the final parallelized loop is given in Figure 10. In the figure, each dimension spans time t at some hyperplanes specified by L and U.

We first explain the process of transformation using the loops in Figure 8(a)-(d). Figure 8(a) shows the general form of a tightly-nested loop. For this loop we choose the shape of the loop body (Section 2.2.3) and replace the original loop body with it, producing Figure 8(b). Here, s_i represents a parallel statement which contains a set of statements, from the original loop body, whose $OFFSET$ values are all i. The maximum $OFFSET$ value is represented by $N_{n+1}-1$ as before. After that, we transform the new loop body into another loop using a case statement, as shown in Figure 8(c). We used i_{n+1} as the index variable for the new innermost loop. This way we can represent the coordinate of each parallel statement instance in $(n+1)$ dimensional space. For example, $(i_1, i_2, \ldots, i_n, i_{n+1})$ is the coordinate of a parallel statement instance which resides at iteration $I(i_1, \ldots, i_n)$ and has $OFFSET$ value i_{n+1}. Then, using the formula in Equation 6, the derivation of which follows shortly, we get an intermediate parallel form as given in Figure 8(d). This form can be improved further to the final form in Figure 10.

Now, our task is to determine L_j and U_j $(1 \leq j \leq n+1)$ in Figure 8(d). But before that, we need to introduce an equation that calculates the cycle of any given parallel statement instance. Given the coordinate of a parallel statement instance, $(i_1, \ldots, i_n, i_{n+1})$, we represent its scheduled cycle as $T(i_1, \ldots, i_n, i_{n+1})$. Since all statement instances in this parallel statement instance reside at iteration $I(i_1, \ldots, i_n)$ and have $OFFSET$ values i_{n+1},

$$T(i_1, \ldots, i_{n+1}) = i_1 \times d_1 + \ldots + i_n \times d_n + i_{n+1}, \tag{5}$$

from Equation 2.

We first consider L_1 and U_1. Imagine an $(n+1)$ dimensional space of parallel statement instances. In this space, we can partition the set of parallel statement instances into N_1 groups, where the l_{th} group contains all parallel statement instances whose coordinates are in the form of $(l, i_2, i_3, \ldots, i_n, i_{n+1})$. Since the starting coordinate of the l_{th} group is $(l, 0, 0, \ldots, 0)$, and the ending coordinate $(l, N_2-1, N_3-1, \ldots, N_{n+1}-1)$, from Equation 5, this group starts at

```
for i_1 = 0 to N_1 - 1
   for i_2 = 0 to N_2 - 1
      ...
      for i_n = 0 to N_n - 1
         x_0
         ...
         x_s
      endfor
   endfor
endfor.
```

(a) The general form of a tightly-nested loop.

```
for i_1 = 0 to N_1 - 1
   for i_2 = 0 to N_2 - 1
      ...
      for i_n = 0 to N_n - 1
         s_0
         ...
         s_{N_{n+1}-1}
      endfor
   endfor
endfor.
```

(b) After shaping.

```
for i_1 = 0 to N_1 - 1
   for i_2 = 0 to N_2 - 1
      ...
      for i_n = 0 to N_n - 1
         for i_{n+1} = 0 to N_{n+1} - 1
            case i_{n+1} is
               0: s_0
               1: s_1
               ...
               N_{n+1} - 1: s_{N_{n+1}-1}
            endcase
         endfor
      endfor
   endfor
endfor.
```

(c) The loop body is transformed into another loop.

```
for t = 0 to MAXCYCLE
   forall i_1 = L_1 to U_1
      forall i_2 = L_2 to U_2
         ...
         forall i_{n+1} = L_{n+1} to U_{n+1}
            case i_{n+1} is
               0: s_0
               1: s_1
               ...
               N_{n+1} - 1: s_{N_{n+1}-1}
            endcase
         endfor
      endfor
   endfor
endfor.
```

(d) The intermediate parallel form.

Figure 8: The process of transforming the loops.

cycle ld_1 and ends at cycle $ld_1 + (N_2 - 1)d_2 + (n_3 - 1)d_3 + \ldots + (N_n - 1)d_n + (N_{n+1} - 1)$.

We want to find groups which contain those parallel statement instances scheduled at cycle t. This problem can be portrayed as in Figure 9(a).

In the figure, each rectangle corresponds to each group. We can see each group is delayed by d_1. Since the heights of the rectangles are all equal, and they are delayed by a fixed amount, d_1, we can calculate those groups which span cycle t. They are from $L_1 = MAX(0, \lceil (t - T(0, N_2 - 1, N_3 - 1, \ldots, N_{n+1} - 1))/d_1 \rceil)$ to $U_1 = MIN(N_1 - 1, \lfloor t/d_1 \rfloor)$.

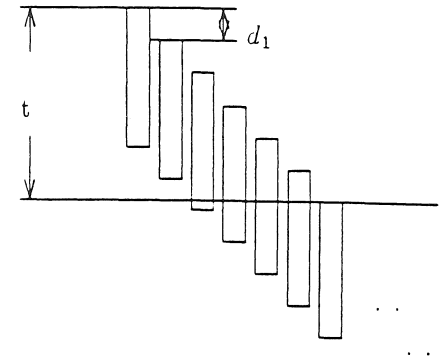
Now, we further partition each of above groups into N_2 groups, where the l_{th} group contain all statement instances whose coordinates are in the form of $(i_1, l, i_3, i_4, \ldots, i_{n+1})$. Here i_1 is an integer between L_1 and U_1. For these groups, we have to identify those which span cycle t. Of these, the first group starts at $T(i_1, 0, 0, \ldots, 0)$ and ends at $T(i_1, 0, N_3 - 1, N_4 - 1, \ldots, N_{n+1} - 1)$. Therefore, proceeding as before, $L_2 = MAX(0, \lceil (t - T(i_1, 0, N_3 - 1, N_4 - 1, \ldots, N_{n+1} - 1))/d_2 \rceil)$ and $U_2 = MIN(N_2 - 1, \lfloor (t - T(i_1, 0, 0, \ldots, 0))/d_2 \rfloor)$. We give a picture to describe this process in Figure 9(b).

In general, suppose we have calculated all L_j and U_j up to $j = k-1$. This means that only those parallel statement instances whose coordinates are in the form of $(i_1, i_2, \ldots, i_{j-1}, i_j, \ldots, i_{n+1})$, with $L_j \leq i_j \leq U_j, \forall j < k$, may be executed at cycle t. Other statement instances which do not have this form can never be executed at cycle t. We take one of the candidate groups, which starts from $(i_1, i_2, \ldots, i_{k-1}, 0, 0, \ldots, 0)$ ending at $(i_1, i_2, \ldots, i_{k-1}, N_k - 1, N_{k+1} - 1, \ldots, N_{n+1} - 1)$, where $L_j \leq i_j \leq U_j, \forall j < k$. We partition it further into N_k groups as before. The first group of these, then, starts at $(i_1, i_2, \ldots, i_{k-1}, 0, 0, \ldots, 0)$ and ends at $(i_1, i_2, \ldots, i_{k-1}, 0, N_{k+1} - 1, \ldots, N_{n+1} - 1)$. Therefore, as can be seen from Figure 9(c),

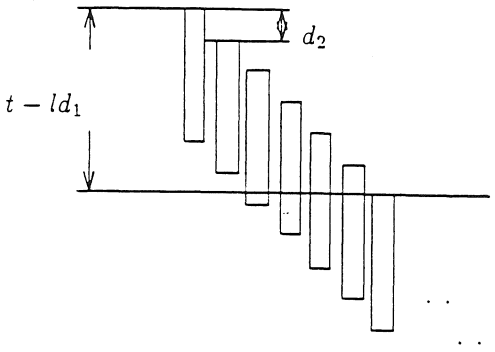
$$
\begin{aligned}
L_k &= MAX(0, \lceil (t - T(i_1, i_2, \ldots, i_{k-1}, 0, N_{k+1} - 1, N_{k+2} - 1, \ldots, N_{n+1} - 1))/d_k \rceil) \\
U_k &= MIN(N_k - 1, \lfloor (t - T(i_1, i_2, \ldots, i_{k-1}, 0, 0, \ldots, 0))/d_k \rfloor)
\end{aligned}
$$

However, if $d_k = 0$, the loop is vectorizable for the k_{th} dimension. Revising the formula with this in mind,

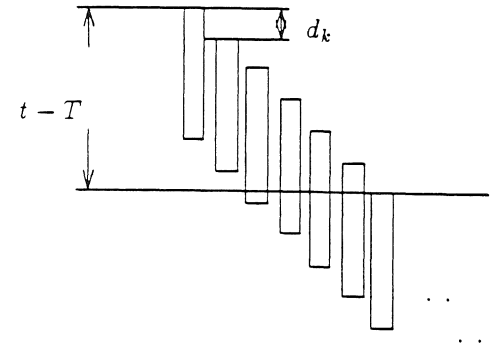
$$
L_k = 0 \text{ (if } d_k = 0), \text{ or } MAX(0, \lceil (t - T_1)/d_k \rceil) \text{ (otherwise)}
$$



(a) Calculating L_1 and U_1.Each rectangle represents an n-dimensional plane.That is, the l_{th} one contains all statement instances whose coordinate fall between $(I(l, 0, 0, \ldots, 0)$ and $I(l, N_2 - 1, \ldots, N_{n+1} - 1)$.



(b) Calculating L_2 and U_2. We have expanded the l_{th} rectangle of the above picture.



(c) Calculating L_k and U_k. We have repeated the process of expansion as in (b) k times.
$T = i_1 d_1 + \ldots + i_{k-1} d_{k-1}$.

Figure 9: Calculation of the delay vector.

$$U_k \quad = \quad N_k - 1 \text{ (if } d_k = 0\text{), or } MIN(N_k - 1, \lfloor (t - T_2)/d_k \rfloor) \text{ (otherwise),} \qquad (6)$$

where

$$T_1 = T(i_1, i_2, \ldots, i_{k-1}, 0, N_{k+1} - 1, N_{k+2} - 1, \ldots, N_{n+1} - 1),$$

and

$$T_2 = T(i_1, i_2, \ldots, i_{k-1}, 0, 0, \ldots, 0).$$

In the above formula, we have defined $d_{n+1} = 1$ to make the formula work for $k = n + 1$.

One thing to note is that in Figure 8(d), $L_{n+1} = U_{n+1}$, always. We informally prove it as follows. Since $d_{n+1} = 1$, from Equation 6,

$$L_{n+1} = MAX(0, (t - T(i_1, i_2, \ldots, i_n, 0),$$

and

$$U_{n+1} = MIN(N_{n+1} - 1, (t - T(i_1, i_2, \ldots, i_n, 0).$$

Since $t = T(i_1, i_2, \ldots, i_n, i_{n+1})$, $t - T(i_1, \ldots, i_n, 0) = i_{n+1}$. i_{n+1} is always between 0 and $N_{n+1} - 1$ inclusive. Therefore $L_{n+1} = U_{n+1} = i_{n+1}$.

Because of this, we can omit the innermost loop in Figure 8(d) and substitute L_{n+1} instead of i_{n+1} in the case statement. Summarizing all above results, we get the final form of parallel loop as in Figure 10.

The user can now verify the derivation of the parallel loop in Figure 2(c) from the sequential one in Figure 2(a). As explained in Section 1, $d_1 = 2$ and $d_2 = 1$ for this loop.

3 Comparison with the wavefront method

We will compare our method with those in [Wolf89], [Bane90], and [LaWo90]. These three all deal with non-vectorizable tightly-nested loops. These methods are very efficient at the iteration level; however, since our approach is geared to the fine grain parallel machines, we are trying to exploit parallelism at all levels – at the statement level as well as at the iteration level. We show the advantage of our fine grain approach. Furthermore, we will point out that our method performs well even when applied at iteration level.

```
for t = 0 to MAXCYCLE
    forall i_1 = L_1 to U_1
        forall i_2 = L_2 to U_2
            . . .
                forall i_n = L_n to U_n
                    case L_{n+1} is
                        0: s_0
                        1: s_1
                        . . .
                        N_{n+1} - 1: s_{N_{n+1}-1}
                    endcase
                endfor
        endfor
    endfor
endfor.
```

$$MAXCYCLE = (N_1 - 1) \times d_1 + \ldots + (N_n - 1) \times d_n + (N_{n+1} - 1)$$

$$L_k = \begin{cases} 0 & \text{if } d_k = 0 \\ MAX(0, \lceil (t - T_1)/d_k \rceil) & \text{otherwise} \end{cases}$$

$$U_k = \begin{cases} N_k - 1 & \text{if } d_k = 0 \\ MIN(N_k - 1, \lfloor (t - T_2)/d_k \rfloor) & \text{(otherwise)} \end{cases}$$

where $T_1 = T(i_1, i_2, \ldots, i_{k-1}, 0, N_{k+1} - 1, N_{k+2} - 1, \ldots, N_{n+1} - 1)$, and $T_2 = T(i_1, i_2, \ldots, i_{k-1}, 0, 0, \ldots, 0)$.

Figure 10: The final parallel form.

[Wolf89] shows an elegant way of implementing the wavefront method. Given a set of skewing factors, a tightly-nested non-vectorizable loop can be parallelized through the combination of loop skewing and loop interchanging. Computing the set of skewing factors roughly corresponds to computing the delay vector in our case. Since this method works at iteration level, the parallelism inside each iteration is ignored. Furthermore because of the way this algorithm exposes parallelism, it ignores the parallelism inside the innermost loop. Thus, even though the innermost loop may be vectorizable, the algorithm sequentializes it.

We will explain the latter point using the example shown in Figure 11(a)-(d). Figure 11(a) is the source code, and Figure 11(b) is the skewed loop with skewing factor = 1. Figure 11(c) shows the interchanged loop with new loop bounds. For this parallel loop, the total number of steps needed for completion is $N_i + N_j - 1$. However, we can simply convert the original innermost loop into a forall loop, reducing the total number of steps needed to N_j, as can be seen in Figure 11(d). Whenever there is a vectorizable loop in the innermost position, this loss of parallelism will result. One remedy is bringing a sequential loop to the innermost position. In the above example, we may bring the i loop to the innermost position before applying loop skewing. However, when the nest depth of the loop gets deeper, identifying a sequential loop and bringing it to the innermost position is not an easy problem.

[Bane90] presents an improved version of the wavefront method for the 2 dimensional case.

```
for i = 1 to N_i
    for j = 1 to N_j
        S_{i,j} = f(S_{i-1,j})
    endfor
endfor
```

(a) The source code.

```
for i = 1 to N_i
    for j = 1 + i to N_j + i
        S_{i,j-i} = f(S_{i-1,j-i})
    endfor
endfor
```

(b) After Loop Skewing.

```
for j = 2 to N_i + N_j
    forall i = max(1, j - N_j) to min(j - 1, N_i)
        S_{i,j-i} = f(S_{i-1,j-i})
    endfor
endfor
```

(c) Parallelized loop.

```
for i = 1 to N_i
    forall j = 1 to N_j
        S_{i,j} = f(S_{i-1,j})
    endfor
endfor
```

(d) More effecient parallel form.

Figure 11: Loss of parallelism in Loop Skewing.

This method concentrates on finding the optimal transform matrix. Once it is found, the loop can be parallelized exposing maximum parallelism at the iteration level. However, The algorithm searches for the optimal transform matrix basically in an exhaustive way, which is expensive even for 2 dimensional case, and prohibitive for the n dimensional case.

[LaWo90] presents another approach to parallelizing non-vectorizable nested loops. Their algorithm consists of two steps. It first converts the original loop nest into a canonical form in which the outermost m loops are fully permutable. Then, the canonical form is transformed into a parallel loop. The second step is mechanical and easy. The first step, however, is expensive, especially as we try to maximize m. [LaWo90] provides a heuristic to maximize m. In the context of fine grain parallelism, our technique has the advantage of concentrating the parallel loops and, in particular, the operation-level parallelism at the innermost nesting levels. Thus, our transformed loops should map directly onto VLIW/superscalar machines and multiprocessors built on top of them.

Now, we take an example to show the advantage of the fine grain approach over the above three methods. See Figure 12(a)-(c). Figure 12(a) is the example loop. Calculating the delay vector for this loop, we get $d_1 = d_2 = 0$, and $d_3 = 2$. The $OFFSET$ values chosen are $OFFSET(A) = 0$, and $OFFSET(B) = 1$. The dimensionality is 3, and $N_4 - 1$ = maximum $OFFSET$ value = 1. Also, $MAXCYCLE = (N_3 - 1)2 + 1 = 2N_3 - 1, L_1 = L_2 = 0, U_1 =$

$N_1 - 1$, and $U_2 = N_2 - 1$. Furthermore, $L_3 = MAX(0, \lceil (t - T_1)/2 \rceil) = MAX(0, \lceil (t-1)/2 \rceil)$, and $U_3 = MIN(N_3 - 1, \lfloor (t - T_2)/2 \rfloor) = MIN(N_3 - 1, \lfloor t/2 \rfloor)$, since $T_1 = T(i_1, i_2, 0, 1) = 1$, and $T_2 = T(i_1, i_2, 0, 0) = 0$. Finally $L_4 = MAX(0, (t - T_1)/1) = MAX(0, t - 2i_3)$. Substituting these values into the parallel form given in Figure 10, we get Figure 12(b).

Applying the technique in [Wolf89] to this loop, we get Figure 12(c). The method in [Bane90] is not applicable since the dimensionality is 3. The method in [LaWo90] will produce the same result as in Figure 12(c) because the loop nest is already fully permutable, and their parallelizing technique for fully permutable loop nests is equivalent to Wolfe's technique.

The total number of steps needed to complete the loop in our case is $2N_3$ (see Figure 12(b)). In the case of Loop Skewing, it is $2N_1 + 2N_2 + 2N_3 - 4$ (See Figure 12(c)). Note that in Figure 12(b), the loop body contains only one statement, either A or B depending on the value of L_4, while in Figure 12(c), the loop body contains two statements. The speed-up is significant; our method removed the first and second dimension completely from the cost. In fact, in our case, the first and second dimensions are vectorizable ($d_1 = d_2 = 0$), which is possible because the scheduling is done at statement level. The iteration level approach in the above three method cannot detect this parallelism. They need to enforce a delay for the first and second dimension, thus leaving the N_1 and N_2 terms in the cost.

Our method also performs well as an iteration level algorithm. Suppose a nested loop in which the loop body contains only a single statement. Even though there is no additional fine grain parallelism, still we see superior performance by our method. The basic reason is that our method does not sequentialize any loop unnecessarily; it simply tries to maximize the overlapping of iterations by minimizing the delay vector. Let's look at the simple example in Figure 11(a) again. The techniques in [Wolf89][LaWo90] produce a parallel loop in Figure 11(c); the total number of steps is $N_i + N_j - 1$ as mentioned before. In our case, since there is only one dependence distance vector, (1,0), the delay vector is $d_i = 1$ and $d_j = 0$; therefore, $MAXCYCLE = (N_i - 1)$. This means the total number of steps in our case is N_i.

4 Conclusion

In this paper, we have shown a new parallelization technique for non-vectorizable tightly-nested loops. Our technique finds an efficient partial ordering (*shape*) of the loop body and

```
for i1 = 0 to N1 − 1
  for i2 = 0 to N2 − 1
    for i3 = 0 to N3 − 1
      A: A_{i1,i2,i3} = f(B_{i1,i2,i3−1})
      B: B_{i1,i2,i3} = g(A_{i1,i2−1,i3}, A_{i1−1,i2−1,i3−1})
    endfor
  endfor
endfor
```

(a) The example loop.

```
for t = 0 to 2N3 − 1
  forall i1 = 0 to N1 − 1
    forall i2 = 0 to N2 − 1
      forall i3 = MAX(0, ⌈(t − 1)/2⌉) to MIN(N3 − 1, ⌊t/2⌋)
        case MAX(0, t − 2i3) is
          0: A_{i1,i2,i3} = f(B_{i1,i2,i3−1})
          1: B_{i1,i2,i3} = g(A_{i1,i2−1,i3}, A_{i1−1,i2−1,i3−1})
        endcase
      endfor
    endfor
  endfor
endfor
```

(b) Parallelization by our method.

```
for i3 = 0 to N1 + N2 + N3 − 3
  forall i1 = MAX(0, i3 − N2 − N3 + 2) to MIN(N1 − 1, i3)
    forall i2 = MAX(0, i3 − i1 − N3 + 1) to MIN(N2 − 1, i3 − i1)
      A: A_{i1,i2,i3−i1−i2} = f(B_{i1,i2,i3−i1−i2−1})
      B: B_{i1,i2,i3−i1−i2} = g(A_{i1,i2−1,i3−i1−i2}, A_{i1−1,i2−1,i3−i1−i2−1})
    endfor
  endfor
endfor
```

(c) Parallelization by the wavefront method

Figure 12: Comparing our method with wavefront methods.

overlaps this shape across all dimensions detecting parallel operations. Three problems are addressed: finding an efficient *shape*, maximizing the overlapped loop area, and expressing the parallelism obtained by this overlapping.

We suggested the *scheduled-order* method for the *shaping* problem, which returns a good *shape* quickly. We found that the overlapping of statements can be maximized by minimizing the *delay vector*, and provided a simple algorithm to calculate it. Finally, we showed a mechanical way of transforming a nested loop into a parallel one when the *delay vector* is known.

Our method has two strong points: it works at fine grain level, and it exploits parallelism from all dimensions. Since it works at statement level, it frequently finds vectorizable dimensions that are not visible by iteration level approaches. And, since it exploits parallelism from all dimensions, even when there is only one statement in the loop body (which means no fine grain parallelism is available), it still performs well compared to previous wavefront methods. We have provided examples to support these arguments. Lastly, since our method parallelizes loops at statement level, it is easy to generalize the algorithm for loosely-nested loops. In the future, we will report about this extension.

References

[AbuS78] Abu-Sufah, W.A., *Improving the performance of Virtual Memory computers,* Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 78-945, (UMI 79-15307), Nov. 1978.

[AiNi88a] Aiken, A. and Nicolau, A. 1988. Optimal loop parallelization. In Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation, June.

[AlKe84] Allen, J.R. and K. Kennedy, *Automatic Loop Interchange*, Proc. of the ACM SIGPLAN'84 Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984; SIGPLAN Notices, Vol. 19, No.6, pp. 233-246, June, 1984.

[Bane88] Banerjee, U., *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[Bane90] Banerjee, U., *Unimodular Transformations of Double Loops*, 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, California, August, 1990.

[CCK87] Callahan, D., Cocke, J., and Kennedy, K., *Estimating Interlock and Improving Balance.*, Proc. of the 1987 International Conf. on Parallel Processing, pp 295-304, August, 1987.

[Cytr84] Cytron,R.G., *Compile Time Scheduling and Optimization for Asynchronous Machines*, Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of comp. sci. Rpt. No. 84-1177,(UMI 85-02121), Oct. 1984.

[Cytr86] Cytron,R.G., *Doacross: Beyond Vectorization for Multiprocessors.* Proc. of the 1986 International Conf. on Parallel Processing, St. Charles, Ill, pp 836-844, August, 1986.

[Fish79] Fisher, J.A. *The Optimization of Horizontal Microcode within and beyond Basic Blocks: an Application of Processor Scheduling with Resources.* Ph.D. thesis, New York Univ., 1979.

[GrLa86] Gross, T. and Lam, M., *Compilation for a high-performance systolic array.*, Proc. for the SIGPLAN 1986 Symposium on Compiler Construction, July 1986.

[Kenn80] Kennedy, K., *Automatic Translation of Fortran Programs to Vector Form*, Rice Technical Report 476-029-4, Rice University, Houston, Oct. 1980.

[KKPL81] Kuck,D.J., Kuhn, R.H., Padua, D.A., Leasure, B., and Wolfe, M., *Dependence Graphs and Compiler Optimization*, Proc of the 8th ACM Symp on Programming Languages, Williamsburg, VA, pp. 207-218, Jan. 1981.

[Kuhn80] Kuhn, R.H., *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage networks, and Decision Trees,* Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 80-1009, (UMI 80-26541), Feb. 1980.

[Lam87] Lam, M., *A Systolic Array Optimizing Compiler.* Ph.D. thesis, Carnegie Mellon Univ., 1987.

[Lamp74] Lamport, L., *The Parallel Execution of DO Loops*, Comm. of the ACM, pp.83-93, Feb. 1974.

[LaWo90] Lam, M. and Wolf, M., *Maximizing Parallelism Via Linear Loop Transformations*, 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, California, August, 1990.

[MuSi87] Munshi, A.A. and Simons, B. *Scheduling Sequential Loops on Parallel Processors.* Proc. of the 1987 International Conf. on Parallel Processing. St. Charles, Ill, August, 1987.

[Nico85] Nicolau, A., *Uniform Parallelism Exploitation in Ordinary Programs.* Proc. International Conf. on Parallel Processing, August 1985.

[Nico87] Nicolu, A., *Loop Quantization or Unwinding Done Right.* Proc. Supercomputing 1st International Conference, June 1987.

[Padu79] Padua, D. *Multiprocessors: Discussions of some Theoretical and Practical Problems,* Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 79-990, Nov. 1979.

[Poly87] Polychronopoulos, C.D., *Loop Coalescing: A Compiler Transformation for Parallel Machines*, Proc. of the 1987 Int'l Conf. on Parallel Processing, Sahni(ed.), Penn State Univ. Press, University Park, PA, pp.235-242, Aug. 1987.

[SDX87] Su,B., Ding, S., and Xia,J., *GURPR – a method for global software pipelining.* In proc. of the 19th annual workshop on microprogramming, pp. 104-108, Oct., 1986.

[Tarj72] Tarjan, R., *Depth-First Search and Linear Graph Algorithms*, SIAM J. Comput., Vol. 1, No. 2, pp146-160, June, 1972.

[Wolf82] Wolfe, M. J., *Optimizing Supercompilers for Supercomputers,* Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 82-1105, Oct. 1982.

[Wolf89] Wolfe,M.J., *Optimizing Supercompilers for Supercomputers*, pp. 136-141, Pitman, London, 1989.